InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees

Conference or Workshop Item

InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees

Nghi D. Q. Bui*, Yijun Yu†, Lingxiao Jiang*,
*School of Computing & Information Systems, Singapore Management Univerity
{dqnbui.2016, lxjiang}@smu.edu.sg
†School of Computing & Communications, The Open University, UK
{y.yu}@open.ac.uk

***Abstract*—Learning code representations has found many uses in software engineering, such as code classification, code search, comment generation, and bug prediction, etc. Although representations of code in tokens, syntax trees, dependency graphs, paths in trees, or the combinations of their variants have been proposed, existing learning techniques have a major limitation that these models are often trained on datasets labeled for specific downstream tasks, and as such the code representations may not be suitable for other tasks. Even though some techniques generate representations from unlabeled code, they are far from being satisfactory when applied to the downstream tasks. To overcome the limitation, this paper proposes *InferCode*, which adapts the self-supervised learning idea from natural language processing to the abstract syntax trees (ASTs) of code. The novelty lies in the training of code representations by predicting *subtrees* automatically identified from the contexts of ASTs. With InferCode, subtrees in ASTs are treated as the labels for training the code representations without any human labelling effort or the overhead of expensive graph construction, and the trained representations are no longer tied to any specific downstream tasks or code units.**

We have trained an instance of InferCode model using Tree-Based Convolutional Neural Network (TBCNN) as the encoder of a large set of Java code. This pre-trained model can then be applied to downstream unsupervised tasks such as code clustering, code clone detection, cross-language code search, or be reused under a transfer learning scheme to continue training the model weights for supervised tasks such as code classification and method name prediction. Compared to prior techniques applied to the same downstream tasks, such as code2vec, code2seq, ASTNN, using our pre-trained InferCode model higher performance is achieved with a significant margin for most of the tasks, including those involving different programming languages. The implementation of InferCode and the trained embeddings are available at the link: https://github.com/bdqnghi/infercode.

I. Introduction

Learning code representations (a.k.a. embeddings) and building a prediction model for programs have been found useful in many software engineering tasks, such as classifying program functionality [1, 2], code search [3, 4, 5], code comment generation [6, 7, 8], predicting bugs [9, 10], translating programs [11, 12], etc. While offering promising performance for the tasks, the prior learning techniques have two major limitations that hinder their performance and generalizability.

- Most code representation models are trained through (semi-)supervised learning. Humans need to manually label the data for a specific downstream task, then engineer features of intermediate representations, and train the models specifically for the task. Such labelling, feature engineering, and training efforts are specific to one particular task and may not be easily transferred to other tasks.
- Even though there are techniques [8, 13] aiming to produce code representations that are transferable to different tasks, their trained code representations are only for some fixed units of code, such as tokens, statements, and functions, and are not flexible to produce embeddings for varying code units. Such techniques may miss useful information across different kinds of code units, and the trained representations may not perform well for various downstream tasks either. Some other techniques based on graph embeddings [14, 15] share a similar drawback and in addition need the overheads of graph construction which may introduce inaccurate information in the graphs.

Such limitations have been illustrated in a recent study: Kang et al. [16] show that the pre-trained code2vec [8] representation does not perform well for other tasks when it was trained specifically for the method-name prediction task.

Towards addressing the limitations, **the aim** of this paper is to develop a new technique for learning code representations, and it should be: (1) trainable without any manual human labeling, (2) flexible in producing embeddings for any code unit that can be parsed into syntax trees, and (3) general enough so that its trained representations for code can perform well for various downstream tasks.

We have two pillars that support the realization of our aim. One is the large amount of source code available on public code hosting platforms, such as Github, Bitbucket, Gitlab. Although the code often lacks accurate labels for downstream tasks, the syntax of the code itself can be checked relatively easily by parsers. It is desirable to leverage such unlabeled data to pretrain the code representations reusable for building various program prediction models for downstream tasks.

The second pillar is supported by the advances of self-supervised learning in machine learning [17, 18, 19, 20, 21]. Such techniques enable the training of neural networks without the need for human labels. Usually, a self-supervised learning technique reformulates an unsupervised learning problem as a supervised one by *automatically generating virtual labels from existing (unlabeled) data*. The self-supervised task, also known as a *pretext task*, guides us to a supervised loss function.

While minimizing the loss function for the pretext task, the technique also produces intermediate representations for the data corresponding to the virtual labels. Because the pretext task can be trained using any data, it is expected that such representations can carry good information about the diverse data and be beneficial to a variety of downstream tasks. This notion of self-supervised learning is very suitable for our aim. Little effort has been invested in the literature to exploit the uses of self-supervised learning for code representation learning. Although some recent work, such as [19], presents a self-supervised learning paradigm for program repair, it is designed specifically for this specific task.

Our key idea is thus to train a pretext task suitable for any source code. Unlike self-supervised learning in natural language processing and visual learning areas that use words or object regions as labels, we utilize the fact that it is relatively easy to obtain the abstract syntax tree (AST) of any syntactically valid code snippets via parsers and it is also easy to identify all the subtrees in ASTs, and automatically use each subtree as the label for the pretext task to predict the probability of that subtree appearing in a particular AST[1]. Fig. 1 illustrates this intuition with an example. The two code snippets implement the same functionality, i.e. bubble sort. If we view these two code snippets as two ASTs, there are many similar subtrees between them. For example, the subtree that represents the conditional expression `arr[j] > arr[j+1]` of the left snippets is similar to `arr[i] > arr[i+1]` although the textual information is quite different. This means that if one can exploit such information, there is no longer the need for labels to build a representation learning model for source code. Unlike the recent uses of neural document embedding models (e.g., doc2vec [22, 23]) for source code (e.g., [24, 25, 26, 27]), our technique learns subtrees in ASTs without the overheads and losses of accuracy in constructing customized graphs from code tokens and node types, although we are also inspired by the same idea of doc2vec. We also provide an alternative to graph-based [28, 29] or execution traces-based [30] embedding techniques as we believe ASTs are more readily available for all kinds of programming languages and may have contained all the code information (although some are hidden).

Based on the key idea, we propose **InferCode**, a self-supervised learning technique for source code by predicting syntax subtrees. As far as we know, we are the first to apply the notation of self-supervised learning to syntax subtrees and can produce code representations for any syntactically valid code snippet without the need of human labelling:

- InferCode can serve as an *encoder* that maps any parsable code snippet into a vector representation (embedding), and this vector can be used for various downstream tasks, such as code clustering, clone detection, and code search.
- InferCode can serve as a pre-trained model and its weights can be reused in downstream training of the models for supervised learning tasks, which can speed up the training and alleviate the issue of lacking data for a particular task.
- We implement InferCode on top of the ASTs produced by SrcML [31] and efficient parsers such as fAST [32]. It provides a combined vocabulary of AST node types for multiple programming languages (e.g., Java, C, C++, C#, Objective C), which implies that our InferCode can be polyglot, producing code representations suitable for tasks involving different languages, such as cross-language code search, as long as the ASTs for a code snippet can be recognized by the parser.

[1]An underlying assumption is that, for such trained representations to capture code meanings, code snippets with the same semantics should involve some syntactically similar code elements. Even though two pieces of code implementing the same functionality can be syntactically different, there could still be some fine-grained elements in the code or other pieces of code that use these two that are syntactically similar, especially when the code base is large.

```
void bubbleSort(int arr[]){
    int n = arr.length;
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
}
```

```
public void bubblesort(int[] array){
    boolean sorted = false;
    while(!sort){
        sorted=true;
        for(int i = 0; i< array.length - 1; i++){
            if(a[i] > a[i+1]){
                int temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
                sorted = false;
}}}}
```

Fig. 1. Example of two code snippets that implement bubble sort in Java that share similar fine-grained code elements.

We have trained an instance of InferCode based on a large set of Java code and evaluated the usefulness of the pretrained code representations in five downstream tasks, three of which are unsupervised (code clustering, code clone detection via similarity measurement, cross-language code search, two are supervised (code classification and method name prediction). For the three unsupervised tasks, we utilize the vectors produce by InferCode and different vector similarity metrics to achieve the goal of each task: For *code clustering*, our results using InferCode outperform the best baseline (Code2vec) by 12% in term of Adjusted Rand Index; For *code clone detection*, our results outperform the best baseline (Code2vec) by 15% in term of F1 score; For *cross-language code search*, our results outperform the best baseline (CLIR) on 13% (on average for multiple languages setting) in term of Mean Reciprocal Rank. For the two supervised tasks, we utilize the weights of the pre-trained model from InferCode to fine-tune the specific prediction model for each task: our results using the fine-tuning process increases the performance of TBCNN for *code classification* by 4% in term of accuracy, which is comparable to ASTNN, the state-of-the-art model for code classification, and increase the performance TBCNN for *method name prediction* by 8%, which is comparable to code2seq, a state-of-the-art model for method name prediction.

II. Related Work

Self-Supervised Learning has made great progress recently for visual data [33, 34, 35, 36, 37, 38]: Gidaris et al. [34] proposed a method to generate different viewpoints of an image by a number of rotations on certain degrees at random and formulate the learning part as a multi-class classification problem over the rotations. This pretext task drives the model

to learn semantic concepts of objects as the parameters of the CNN image encoder; Zhang et al. [35] proposed to use colorization as the pretext task by giving colours to a grayscale input image in order to map this image to a distribution over quantized color value outputs.

There has been tremendous effort in exploring self-supervised learning in Natural Language Processing (NLP) research [22, 23, 39, 40, 41, 42]. Word2vec [22] is a form of self-supervised learning, which aims to learn good representation of words by taking a small chunk of the text of certain window size. Doc2vec [23] shares the same principle with word2vec, which aims to use a document to predict the words inside it so that similar documents will have similar embeddings; Skip-thought vectors [39] builds a statistical language model by predicting the neighbouring sentences of a centering sentence; BERT [40] advances the language models by masking the words in a text randomly in order to predict them.

Deep Code Learning Models: There has been a huge interest in applying deep learning techniques to software engineering tasks such as program functionality classification [43, 44], bug localization [45, 46], function name prediction [47], code clone detection [44], program refactoring [6], program translation [11], and code synthesis [48]. Allamanis et al. [49] extend ASTs to graphs by adding a variety of code dependencies as edges among the tree nodes, intended to represent code semantics, and apply Gated Graph Neural Networks (GGNN) [50] to learn the graphs from code; Code2vec [8], Code2seq [13], and ASTNN [44] are designed based on splitting ASTs into smaller ones, either as a bag of path-contexts or as flattened subtrees representing individual statements. They use various kinds of Recurrent Neural Networks (RNNs) to learn such code representations. Unfortunately, there is little effort in designing the source code model with unlabelled data. Yasunaga and Liang [19] presents a self-supervised learning paradigm for program repair; surveys on code embeddings [25, 27] present evidence to show that there is a strong need to alleviate the demands of labelled data and encourage the community to invest more into the methods for learning source code with unlabelled data.

Our approach differs from existing ways to reuse the pretrained code learning model: Kang et al. [16] reuse the token embeddings from Code2vec for downstream tasks only to find lower performance than simpler word embedding methods like Word2vec. In contrast, we use the weights of the pretrained model and the code vector $\vec{v}$ produced by the encoder instead of the token embeddings.

III. Preliminaries

A. Source Code Representation Learning

Source code representation learning usually contains the following two phases: (1) representing a code snippet into an intermediate representation (IR), such as token streams, ASTs, AST paths or graphs; and (2) designing a neural network suitable to process such intermediate representations. Such a neural network can also be called an *encoder*, which receives the code IR and maps it into a code vector embedding $\vec{v}$ (usually a combination of various kinds of code elements), then $\vec{v}$ can be fed into the next layer(s) of a learning system and trained for an objective function of the specific task of the learning system. For example, in Code2vec [8], $\vec{v}$ is a combination of different AST paths. In GGNN [49] or TBCNN [43], $\vec{v}$ is a combination of AST nodes. A trained model, either on supervised learning or self-supervised learning task, can produce $\vec{v}$. In our work, we will evaluate how the $\vec{v}$ trained on a self-supervised learning objective function over a large set of unlabelled data can be made useful for different downstream SE tasks.

B. Neural Document Embedding Models

Doc2vec [23] is an extension to word2vec [22]. Doc2vec uses an instance of the skip-gram model called paragraph vector, which is a distributed bag of words (interchangeably referred as doc2vec skip-gram) that is capable of learning the representations of a sequence words of arbitrary lengths, such as sentences, paragraphs and even whole documents. More specifically, given a set of documents $\{d_1, d_2, ...d_n\}$ and a sequence of words $\{..., w_{ij}, ...\}$ sampled from the document d_i, skip-gram learns a D-dimensional embeddings of the document d_i and each word w_{ij} sampled, i.e., $\vec{v}_i, \vec{v}_{ij} \in \mathbb{R}^D$, respectively. The model works by considering a word w_{ij} to be occurring in the context of document d_i and tries to maximize the following log likelihood function: $\sum_j log\ Pr(w_{ij}|d_i)$, where the probability $Pr(w_{ij}|d_i)$ is defined as $\frac{exp(\vec{v}_i \cdot \vec{v}_{ij})}{\sum_{w \in \mathcal{V}} exp(\vec{v}_i \cdot \vec{w})}$, where $\mathcal{V}$ is the vocabulary of all the words across all documents.

In this paper, we consider ASTs analogous to documents and subtrees in the ASTs analogous to words in the documents, and adapt the idea of document embedding to learn the embeddings of ASTs of any size by using an encoder for the AST of any parsable code snippets.

C. Self-supervised Learning Formulation

The goal of self-supervised learning is to train an encoder E such that E can map an object into a vector representation (embedding). In our case, the embedding $\vec{v}$ is for the AST representation T of a code snippet C. Training the encoder E is to learn its parameters (or weights) so that E is able to produce the embeddings for the code snippets such that the vectors for the snippets having similar syntactical and semantic information will be close in the vector space. In visual learning, Convolutional Neural Networks (CNNs) are usually chosen as the encoder for images. In NLP, Recurrent Neural Networks, or recently, BERT, is typically used as the encoder for text sequences. In our case, we choose Tree-based CNN as the source code encoder as it has been successfully used before [43, 51, 52, 53] and justified further in Section VIII.

Given a dataset X, for each data X_i in X, there is a corresponding pseudo label P_i automatically generated for a predefined pretext task without involving any human annotation. Given a set of n training data $D = \{P_i\}_{i=1}^n$, the aim is to minimize the loss function: $loss(D) = \frac{1}{n} \sum_{i=1}^n loss(X_i, P_i)$. We can easily identify subtrees in ASTs as the pseudo labels P

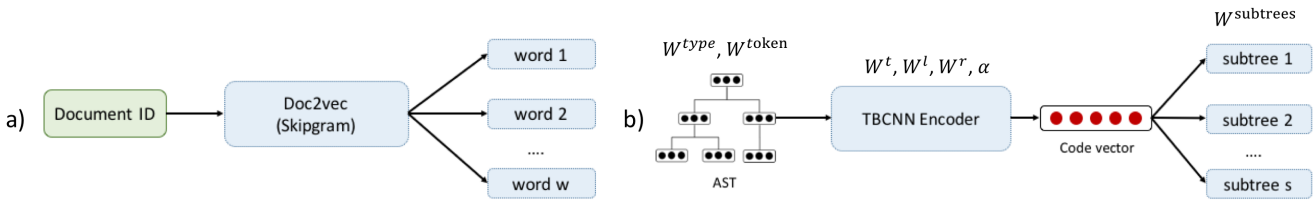

Fig. 2. a) Doc2vec's skipgram model - Given a document d, it samples c words and considers them as co-occurring in the same context of d to learn d's representation; (b) InferCode - Given an AST T, it samples s subtrees from T and uses them as the context to learn T's representation.

automatically without human annotations so that our learning technique can be self-supervised.

IV. Approach Details

A. Overview

Figure 2 presents a high-level view of our InferCode approach as an analogy to Doc2vec by treating an entire AST as a document and treating its subtrees as words in the document. Given a set of ASTs $\{T_1, T_2, ...T_n\}$, and a set of all subtrees $\{..., T_{ij}, ...\}$ of T_i, we represent T_i, T_{ij} by D-dimensional embedding vectors $\vec{v}_i, \vec{v}_{ij} \in \mathbb{R}^D$, respectively. By considering a subtree $T_{ij} \in T_i$ to be occurring in the context of the AST T_i, we aim to maximize the following logarithmic likelihood: $\sum_j log\ Pr(T_{ij}|T_i)$.

Unlike doc2vec, InferCode does not query the embedding vectors directly from an embedding matrix for the whole documents; instead, we first encode the entire AST to obtain the $\vec{v}_i$, then use it to predict the subtrees. The steps of our technique are as follows:

- For each AST in our dataset, we identify a set of subtrees, and all of the subtrees are accumulated into a vocabulary of subtrees (Section IV-B);
- We feed an AST into a Tree-Based CNN (TBCNN) encoder to produce a code vector $\vec{v}_i$. Then $\vec{v}_i$ is used to predict the subtrees identified in the previous step;
- After the encoder has been trained, we can use it as the pretrained model for downstream tasks.

B. Process to Identify Subtrees

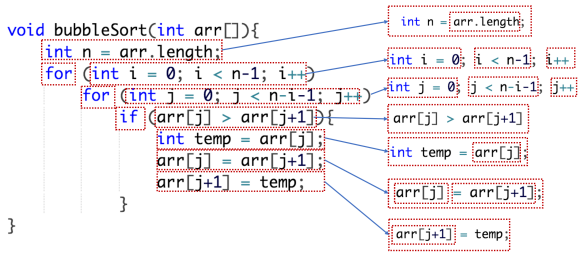

Fig. 3. Example to generate subtrees from a code snippet

By traversing an AST, every visited node satisfying a certain condition, e.g., of the type `expr`, leads to a subtree rooted at the visited node. In our experiments, we chose to select the subtrees whose root node is of the types {`expr_stmt`, `decl_stmt`, `expr`, `condition`}, We consider these relatively fine-grained code elements because they are usually meaningful yet small enough to be considered as frequent "words" in the vocabulary of subtrees from a large code base. Such small code elements often have similar meaning when their syntactical structure is similar even though their textual appearance may be different (due to different identifier names, such as `int n = arr.length` versus `int m = x.length`). In addition, we also consider the nodes that represent for a single keyword, such as `if, for, while`. Noted that these nodes can be seen as the subtrees with size = 1.

We do not consider coarse-grained subtrees such as the whole `if, while, for` statements, as those subtrees are often too big so that (1) each of them, as an individual vocabulary word, may appear too infrequent in the code base for the encoder to learn a meaningful representation for it directly; (2) syntactical differences among the big subtrees do not necessarily mean the corresponding code has different meanings, while the encoder may have harder time to recognize the semantic similarity among them.

Figure 3 shows a sample bubble sort code snippet written in Java and the identified subtrees on the right hand side. This snippet is parsed into an AST, and certain subtrees are identified automatically. For example, the statement `int n = arr.length` contains an expression `arr.length`. Both `int n = arr.length` and `arr.length` are identified.

C. Learning Source Code Representation

Once we have the subtrees, we can use them to learn the source code encoder under a self-supervision mechanism. Here we choose TBCNN [43] as the source code encoder. There are two major differences between our implementation of TBCNN and the original design in [43]: we include the textual information into the node initialization embedding instead of using only the type information, and we replace the dynamic max pooling with an attention mechanism to combine node embeddings. Figure 4 shows an overview of the workflow of the TBCNN with the modifications we made. There are three steps to learn the weights of the encoder, which are described as follows:

- **Learning Nodes Representation**: This step is to learn the representation of the node of the input AST T. The information of the tree will propagate from bottom to top, i.e., a parent node will accumulate the information of its descendant in the AST. After the accumulation step, each node will contain the information of its descendants.
- **Aggregating Nodes Information**: Since we want to represent the AST representation of the code snippet into a

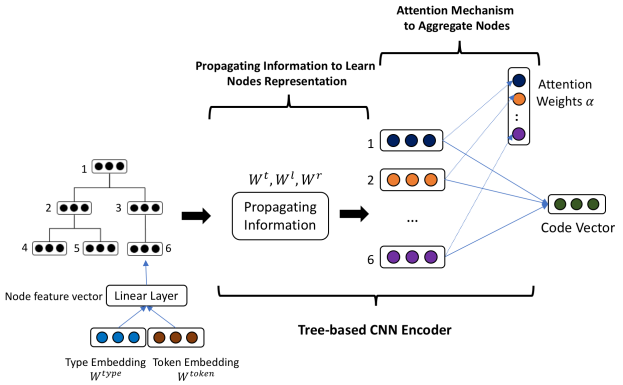

Fig. 4. Workflow of Tree-based Convolutional Neural Network [43] with 2 modifications: (1) including the token information to initialize the node vector; and (2) using the attention mechanism to aggregate node's information

fixed dimension vector $\vec{v}$, we need to combine all the node embeddings into one fixed single embedding. We use the attention layer for this purpose.

- **Predicting Subtrees**: Once having the v_C, we use it to predict the subtrees extracted from T. Intuitively, this process is similar to Eq. (III-B), where the task is to predict the probability of a subtree given the embedding v_C.

1) Learning Nodes Representation with TBCNN: We briefly introduce the Tree-based Convolutional Neural Networks (TBCNN, [43]) for processing AST inputs.

A tree $T = (V, E, X)$ consists of a set of nodes V, a set of node features X, and a set of edges E. An edge in a tree connects a node and its children. Each node in an AST also contains its corresponding texts (or tokens) and its type (e.g., operator types, statement types, function types, etc.) from the underlying code. Initially, we annotate each node $v \in V$ with a D-dimensional real-valued vector $\vec{x_v} \in \mathbb{R}^D$ representing the features of the node. We associate every node v with a hidden state vector $\vec{h_v}$, initialized from the feature embedding $\vec{x_v}$. In [43], the node is initialized only with the type embedding. In our case, we initialize the node with a fusion of the embeddings of its texts and through a linear layer. The embedding matrices for the texts and types are learnable in the whole model training pipeline, formally defined as $\mathbf{W}^{type}$ and $\mathbf{W}^{token}$, respectively.

In TBCNN, a convolution window over an AST is emulated via a binary tree, where the weight matrix for each node is a weighted sum of three fixed matrices $\mathbf{W}^t$, $\mathbf{W}^l$, $\mathbf{W}^r \in \mathbb{R}^{D \times D}$ (each of which is the weight for the "top", "left", and "right" node respectively) and a bias term $\mathbf{b} \in \mathbb{R}^D$ Hence, for a convolutional window of depth d in the original AST with $K = 2^d - 1$ nodes (including the parent nodes) belong to that window with vectors $[\mathbf{x}_1, ..., \mathbf{x}_K]$, where $\mathbf{x_i} \in \mathbb{R}^D$, the convolutional output $\mathbf{y}$ of that window can be defined as: $\mathbf{y} = tanh(\sum_{i=1}^{K}[\eta_i^t \mathbf{W}^t + \eta_i^l \mathbf{W}^l + \eta_i^r \mathbf{W}^r]\mathbf{x}_i + \mathbf{b})$, where $\eta_i^t, \eta_i^l, \eta_i^r$ are weights calculated corresponding to the depth and the position of the nodes.

2) Attention Mechanism to Aggregate Nodes: After the nodes representation has been learned, we need an aggregation method to combine all the nodes in to one fixed embedding that represent for the code snippet. Mou et al. [43] use max pooling to combine the nodes. However, max pooling may discard a lot of important information, so we replace it with the attention mechanism to aggregate nodes. Formally, an attention vector $\vec{a} \in \mathbb{R}^D$ is initialised randomly and learned simultaneously with updates of the networks. Given n node state vectors: $\{\vec{h_1}, ..., \vec{h_n}\}$, the attention weight α_i of each $\vec{h_i}$ is computed as the normalised inner product between the node state vector and the global attention vector: $\alpha_i = \frac{\exp(\vec{h_i}^T \cdot \vec{a})}{\sum_{j=1}^{n} \exp(\vec{h_j}^T \cdot \vec{a})}$. The exponents in this equation are used to make the attention weights positive, and they are divided by their sum to have a max value of 1, as done by a standard softmax function.

The aggregated code vector $\vec{v} \in \mathbb{R}^D$ represents the whole code snippet. It is a linear combination of the node state vectors $\{\vec{h_1}, ..., \vec{h_n}\}$ weighted by their attention scores:

$$\vec{v} = \sum_{i=1}^{n} \alpha_i \cdot \vec{h_i} \tag{1}$$

3) Predicting Subtrees: From the process to extract the subtrees, we have a vocabulary of all subtrees from our training dataset. The embeddings of subtrees are learn-able parameters, formally defined as $\mathbf{W}^{subtrees} \in \mathbb{R}^{|L| \times D}$, where L is the set of subtrees extracted from the training corpus. The embedding of $subtrees_i$ is row i of $\mathbf{W}^{subtrees}$. The predicted distribution of the model $q(l)$ is computed as the (softmax-normalized) dot product between the code vector $\vec{v}$ and each of the subtree embeddings:

$$for\, l_i \in L:\ q(l_i) = \frac{\exp(\vec{v}^T \cdot \mathbf{W}_i^{subtrees})}{\sum_{l_j \in L} \exp(\vec{v}^T \cdot \mathbf{W}_i^{subtrees})} \tag{2}$$

where $q(l_i)$ is the normalized dot product between the vector of l_i and the code vector $\vec{v}$, i.e., the probability that a subtrees l_i appears in a given code snippet $\boldsymbol{C}$. This is aligned with Eq. (III-B) in Doc2vec to predict the likelihood of a word given a document.

Finally, we need to learn these parameters of InferCode: $\mathbf{W}^{type}$, $\mathbf{W}^{token}$, $\mathbf{W}^t$, $\mathbf{W}^l$, $\mathbf{W}^r \in \mathbb{R}^{D \times D}, a \in \mathbb{R}^D, \mathbf{W}^{subtrees} \in \mathbb{R}^{|L| \times D}$.

D. Usage of the Model after Training

We have presented the pipeline to train InferCode by predicting subtrees as the labels. Note that in self-supervised learning, one does not usually care about the performance of the pretext task. Instead, we care about the weights that have been learned and the ability of the model to generate the embeddings. The trained TBCNN encoder of InferCode can be used to produce an embedding vector $\vec{v}$ for any parsable code snippet by (1) parsing the code into an AST and (2) feeding the AST through the encoding step presented in Figure 4 to get the vector. The weights in the trained model can also be used for the prediction models in downstream supervised learning tasks to save training costs and potentially improve their prediction accuracy. We illustrate the usages in next sections.

V. Use Cases

In this section, we briefly describe how InferCode can be adapted into 5 different downstream tasks.

A. Code Embedding Vectors for Unsupervised Tasks

1) Code Clustering: The task is to put similar code snippets automatically into the same groups without any supervision. Given the code vectors $\vec{v}$ produced by the pre-trained InferCode for any code snippets, we can realize the task by defining a similarity metric based on Euclidean distance and applying a clustering algorithm such as K-means[54].

2) Code Clone Detection: There are supervised and unsupervised approaches to detect clones. While deep learning methods are applied to detect code clones, they require labelled data to train a supervised learning model [14, 44, 55]. As such, one needs human annotators to mark the pairs of snippets as clones, limiting the ability to detect clones by large amount of the data one can collect.

To alleviate the need of labelled pairwise data to train supervised clone detectors, we opt to use the unsupervised approach based on a good similarity measurement: For a pair of code snippets, we measure the similarity of between the two vectors by using the cosine similarity; when the cosine similarity between the vectors are higher than a certain threshold, we treat the pair as clones. In this work, we choose 0.8 as the threshold.

3) Cross Language Code-to-Code Search: *Code-to-code* search is useful for developers to find other code in a large code base that is similar to a given code query. For example, a developer working on a task to migrate a sorting algorithm implemented in Java to another language (e.g., C#) might want to see if there exists an implementation of the same sorting algorithm in C#, instead of rewriting the code in C# from scratch. Existing code-to-code search engine such as Krugle, Facoy [4], Aroma [56], only consider the searching problem within one programming language. Considering the more challenging cross-language search use case, our pre-trained InferCode model can be more useful. The backbone of InferCode is ASTs, and we used the ASTs from an efficient parser for SrcML representations [32] because it is a combined vocabulary for the AST node types in five mainstream languages (Java, C, C++, C# and Objective C). Our pre-trained model can receive SrcML AST structure of any code snippets within these 5 languages. Given a code snippet in one language as a query, we aim to retrieve other code snippets that are functionally similar to the given code snippet in other programming languages. Since all code snippets can be represented in the form of vector representations, this problem can be formalized as the nearest-neighbor query in the vector space.

B. Fine-Tuning for Supervised Learning Tasks

A paradigm to make use of large amount of unlabelled data is *self-supervised pretraining followed by a supervised fine-tuning* [17, 18], which reuses parts (or all) of a trained neural network on a certain task and continue to train it or simply using the embedding output for other tasks. Such fine-tuning processes usually have the benefits of (1) speeding up the training as one does not need to train the model from randomly initialized weights and (2) improving the generalizability of the downstream model even when only small datasets have labels.

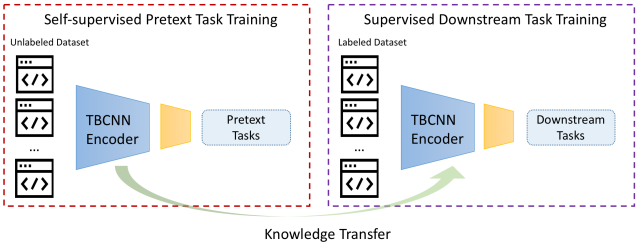

Fig. 5. Code features are learned through the training process of TBCNN encoder to solve a predefined pretext task. After finishing the training, the learned parameters serve as a pre-trained model and can be transferred to other downstream tasks by fine-tuning. The performance on these downstream tasks is used to evaluate the quality of the learned features.

As shown in Figure 5, the TBCNN encoder of InferCode serves as a pretrained model, in which the weights resulted from the self-supervised learning are transferred to initialize the model of the downstream supervised learning task.

1) Code classification: Here we use *code classification* [43] as a downstream task to demonstrate the usefulness of the fine-tuning process. This task is to, given a piece of code, classify the functionality class it belongs to.

2) Method name prediction: We use *Method name prediction* [8] as the second downstream task. This task is to, given a piece of code (without its function header), predict a meaningful name that reflects the functionality of the code.

VI. Empirical Evaluation

In this section, we evaluate InferCode on the five use cases presented in Section V. We want to see to what degree the pre-trained model is applicable to different use cases even when the cases involve multiple programming languages.

For the training phase, we reuse the Java-Large dataset that has been used in Code2vec [8] and Code2seq [13]. This dataset contains a large number of Java projects collected from Github (4 million files). For the testing phase, we use different datasets for each of the task as the *test data*.

We parse all the files into ASTs using fast [32]. Then we identify all the subtrees to form a vocabulary of subtrees. Having the ASTs, and the subtrees as the pseudo labels, we train the InferCode model by using the softmax cross-entropy as the objective loss function and choose Adam [57] as the optimizer with an initial learning rate of 0.001 on an Nvidia Tesla P100 GPU.

A. Code Clustering

1) Datasets, Metrics, and Baselines: We use two datasets for this task. The first is the OJ dataset that contains 52,000 C code snippets known to belong to 104 classes [43]. The second is the Sorting Algorithm (SA) dataset used in [58], which consists of 10 classes of sorting algorithm written in

Java, each algorithm has approximately 1000 code snippets. Our clustering task here is to cluster all the code snippets (without class labels) according to the similarity among the code vectors: For the OJ dataset, we use K-means (K=104) to cluster the code into 104 clusters; For the SA dataset, we use K-means (K=10) to cluster the code. Then we use the class labels in the datasets to check if the clusters are formed appropriately.

We use the Adjusted Rand Index [59] as the metric to evaluate the clustering results. Here we present the definition of Rand Index. Let C be the ground truth class assignment, and K be the number of clusters assigned by a clustering algorithm. Let a be the number of pairs of elements that are in the same set in C and the same set in K; and b as the number of pairs of elements that are in different sets in C and different sets in K. Rand Index for two datasets can be defined as: $RI = \frac{a+b}{\binom{n_{samples}}{2}}$, where the combinatorial number $\binom{n_{samples}}{2}$ is the total number of possible pairs in the dataset (without ordering). However, the RI score does not guarantee that random label assignments will get a value close to zero (esp. if the number of clusters is in the same order of magnitude as the number of samples). To counter this effect, *Adjusted Rand Index* is defined by discounting the expected RI of random labelling as followed: $ARI = \frac{RI - E[RI]}{max(RI) - E[RI]}$.

For the baselines, if we treat source code as text, the self-supervised learning techniques in NLP can also be applied for code. As such, we include two well-known baselines from NLP, Word2vec [22], and Doc2vec [23]. We also include another baseline from [60], a state-of-the-art method to learn sentence representation. This method uses a Sequential Denoising Auto Encoder (SAE) method to encode the text into an embedding, and reconstruct the text from such embedding. We also compare with two baselines for code modeling, Code2vec [8] and Code2seq [13]. Code2vec works by training a path encoder on bag-of-paths extracted from the AST. The path encoder will encode the paths into an embedding $\vec{v}$, then use $\vec{v}$ to predict the method name. Code2seq shares a similar principle, but $\vec{v}$ is used to generate a textual summary of code. In either case, we use the path encoders of Code2vec and Code2seq to produce the code vectors and also perform the same clustering process as InferCode.

2) Results: Table I shows the results of code clustering using different models. InferCode performs the best for both datasets. The NLP methods, however, underperform other code learning methods. This is reasonable because both Code2vec and Code2seq capture structural information from code, while NLP methods treat code as text sequences. We will provide a deeper analysis of the clusters by providing visualizations of the vectors produced by different methods (see Section VII-A).

TABLE I
RESULTS OF CODE CLUSTERING IN ADJUSTED RAND INDEX (ARI)

Model	Performance (ARI)	
	OJ Dataset (C)	SA Dataset (Java)
Word2vec	0.28	0.24
Doc2vec	0.42	0.29
SAE	0.41	0.31
Code2vec	0.58	0.51
Code2seq	0.53	049
InferCode	**0.70**	**0.62**

B. Code Clone Detection

1) Datasets, Metrics and Baselines: We use two datasets in two languages. One is the OJ Dataset again that contains 52,000 C/C++ programs. The other is the BigCloneBench, a Java dataset that has been widely used to benchmark code clone detection techniques, which consists of projects from 25,000 projects, covering 10 functionalities and including 6,000,000 true clone pairs and 260,000 false clone pairs. For the OJ Dataset, we followed the process in Zhang et al. [44] to construct a set of code pairs for clone detection based on pair-wise similarity measurement, so-called OJClone: We choose 500 programs from each of the first 15 programming problems in OJ. It would produce a total of 1.8 million clone pairs and 26.2 million non-clone pairs, which are extremely time-consuming for comparison. So that we randomly select 50000 samples clone pairs and 50000 non-clone pairs for measuring the performance of various clone detectors.

We use the well-known Precision, Recall, and F1 scores. Since the task is unsupervised, in this paper we compare InferCode only with unsupervised clone detectors that do not require labeled data (although the pretrained InferCode can also be applied to supervised clone detection). The baselines include Deckard [61], SourcererCC [62], DLC [63], and a detector using the code vectors extracted from Code2vec [8, 16] and the same cosine similarity threshold used for InferCode.

2) Results: Table II shows the overall precision, recall and F1 for InferCode and other baselines. The detector based on InferCode has the highest recall (except for SourcererCC whose precision is relatively low). Overall in terms of F1, it outperforms other unsupervised clone detectors.

Note that we do not compare with techniques such as Oreo [55], CCD [14], ASTNN [44] because they use supervised learning techniques to build clone *classifiers*. We believe that the code embeddings or the weights from the pretrained InferCode can be used for training supervised clone classifiers too, and with further improvement on self-supervised learning techniques such as improving the encoder, the auto-identified labels, and the loss function, the performance of unsupervised code clone detection may also get close to supervised ones. We leave these evaluations for future work.

TABLE II
RESULTS OF CODE CLONE DETECTION IN PRECISION, RECALL AND F1

Methods	BigCloneBench (Java)			OJClone (C)		
	P	R	F1	P	R	F1
Deckard	0.93	0.02	0.03	0.99	0.05	0.10
DLC	0.95	0.01	0.01	0.71	0.00	0.00
SourcererCC	0.88	0.02	0.03	0.07	0.74	0.14
Code2vec	0.82	0.40	0.60	0.56	0.69	0.61
InferCode	0.90	0.56	0.75	0.61	0.70	0.64

C. Cross Language Code-to-Code Search

1) Datasets, Metrics, and Baselines: Given the implementation of an algorithm in one language, this task is to search for other implementations of the same algorithm written in other languages. So we need a dataset that contains multiple implementations of algorithms in different languages. We construct such a codebase by searching from the Rosetta Code[2] and other code from GitHub: We collect code in Java, C, C++, C# from Rosetta Code which results in around 3000 samples, then we collect 5000 random program files from Github for each of the languages and mix them with the samples.

For instance, for Java, we collect a large set of Java projects from Github that have at least 10 stars. There is a possibility that the collected GitHub projects contain implementations of the algorithms in the Rosetta Code. So we perform a simple text filtering to exclude all the files that contain a token of any of the algorithm name. Let us take 3 algorithms as examples (Bubble-sort, Singly-linked-list-Traversal, Yin-yang[3]): We exclude any file that contains any of these tokens: {*bubble, sort, singly, linked, list, traversal, yin, yang*}. Then for the remaining Java files, we sample a subset of 5000 files and mix them with the Java implementations of the algorithms from the Rosetta dataset. We do the same for C#, C++, C, and obtain in total about 23,000 files in our search code base.

With the constructed code base, we perform the evaluation for cross-language search as follows: For each of the 3000 code files from Rosetta Code, say a bubble sort implementation written in Java, we use it as the query to retrieve other files containing top-K similar code. Here we choose K = 10 in this evaluation. The ideal query results should only return a list of code snippets that are from Rosetta Code but implement the same bubble sort algorithm in C++, C#, and C; other results would be considered as false positives. Since our assumption is that there is only one relevant result for the query, we use the well-known Mean Reciprocal Rank (MRR) as the metric to evaluate the actual query results. This task can be formulated as the information retrieval (IR) problem and the neural IR techniques are widely applied recently for textual data [64, 65, 66], we include Word2vec, Doc2vec, CLIR [66], a cross-lingual information retrieval system for text. We also follow Sachdev et al. [5] to include ElasticSearch, a fuzzy text search baseline. Although there are recent methods designed specifically for code-to-code search, such as Facoy [4] and Aroma [56], they are designed only for monolingual code search, thus we do not compare with them directly.

2) Results: Table III shows the results for InferCode and other baselines. The performance of InferCode is the best among all the models. ElasticSearch, on the other hand, performs the worst; this is expected because ElasticSearch is a simple fuzz text search technique not designed to capture structural information of code.

[2]http://www.rosettacode.org, https://github.com/acmeism/RosettaCodeData

[3]These are taken from the names of the algorithms at https://github.com/acmeism/RosettaCodeData/tree/master/Task

TABLE III
RESULTS OF CROSS-LANGUAGE CODE-TO-CODE SEARCH IN MEAN RECIPROCAL RANK (MRR)

Approach	Performance (MRR)			
	Java	C#	C++	C
ElasticSearch	0.13	0.18	0.22	0.21
Word2vec	0.33	0.36	0.30	0.32
Doc2vec	0.32	0.34	0.38	0.30
CLIR	0.29	0.32	0.34	0.39
InferCode	**0.57**	**0.45**	**0.51**	**0.54**

D. Fine-Tuning for Supervised Learning Tasks

1) Datasets, Metrics, and Baselines:

a) Code Classification: We again use the OJ Dataset for this task. We split this dataset into three parts for training, testing, and validation by the ratio of 70:20:10. Out of the training data, we feed X% to the neural model, where X = 1, 10, 100. We then initialize the neural model either randomly or with the weights from the pre-trained InferCode. Therefore, we have four settings for training the supervised model for comparison: fine-tuning the TBCNN encoder with 1%, 10%, or 100% of the labeled training data respectively, and the randomly initialized model. Using only 1% or 10% is to demonstrate that given a pre-trained model, one only needs a small amount of labeled data to achieve reasonably good performance for the downstream task.

We use the accuracy metric widely used for classification tasks. As the baselines, we include the ASTNN [44] trained from scratch, which is a state-of-the-art model for code classification on the OJ dataset, and TextCNN [67] and Bi-LSTM [68] trained with 100% of the training data, which are widely used for text classification.

b) Method Name Prediction: We use the Java-Small dataset widely used as a benchmark for method name prediction and has been used in Code2vec [8] and Code2seq [13]. This dataset has already been split into three parts, namely training, testing, and validation. We perform the same evaluation protocol as the code classification task by fine-tuning the model with 1%, 10%, and 100% of the labeled training data, in contrast to random initialization of the model without fine-tuning. To predict the method name, we follow Code2vec to use the code vector $\vec{v}$ to predict the embedding of a method name from a lookup table (see Section 4.2 in Code2vec [8]). We measure prediction performance using precision (P), recall (R), and F1 scores over the sub-words in generated names, following the metrics used by Alon et al. [8]. For example, a predicted name `result_compute` is considered as an exact match of the ground-truth name `computeResult`; predicted `compute` has full precision but only 50% recall; and predicted `compute_model_result` has full recall but only 67% precision.

2) Results: Table IV shows the results for code classification. Fine-tuning on 10% of the training data gets comparable results with the NLP baselines. Fine-tuning on 100% of the training data gets comparable with ASTNN, a state-of-the-art model for code classification on the OJ dataset.

TABLE IV
RESULTS OF CODE CLASSIFICATION IN ACCURACY WITH FINE-TUNING (FT) ON THE OJ DATASET

Approach	FT (1%)	FT (10%)	FT (100%)	Supervised
InferCode	70.4%	87.6%	**98.0%**	94%
TextCNN	-	-	-	88.7%
Bi-LSTM	-	-	-	88.0%
ASTNN	-	-	-	97.8%

TABLE V
RESULT OF METHOD NAME PREDICTION IN F1 WITH FINE-TUNING (FT) ON THE JAVA-SMALL DATASET

Approach	FT (1%)	FT (10%)	FT (100%)	Supervised
InferCode	20.31%	30.54%	**43.33%**	35.67%
Code2vec	-	-	-	18.62%
Code2seq	-	-	-	43.02%

Table V shows the results for method name prediction. We get a comparable result with Code2seq when fine-tuning with 100% labeled data.

E. Summary

InferCode outperforms most of the baselines across five tasks, including three unsupervised ones (code clustering, code clone detection via similarity measurement), cross-language code-to-code search), and two supervised ones (code classification and method name prediction).

Note that this does not mean that the TBCNN encoder in InferCode is better than ASTNN, Code2vec, or Code2seq, as those neural models can be used as the encoder in InferCode too. It only means that pre-training a model on large unlabeled data using self-supervised learning to predict subtrees can produce more transferable models while maintaining the performance of such models for various code learning tasks.

The performance of the self-supervised learning models may be improved further with different encoders. We leave those explorations for future work.

VII. ANALYSIS

This section analyses the effects of various parameters on the performance of different tasks.

A. Cluster Visualization

To help understand why the vectors produced by InferCode are better than the vectors produced by others, we visualize the vectors of the programs from the OJ dataset that have been used for the code clustering. We choose the embeddings produced by Doc2vec, Code2vec, and InferCode for the first 9 classes of the OJ dataset, then we use T-SNE [69] to reduce the dimension of the vectors into two-dimensional space and visualize. As shown in Figure 6, (1) the vectors produced by InferCode group similar code snippets into the same cluster with clearer boundaries, and (2) The boundaries among clusters produced by Doc2vec and Code2vec are less clear, which makes it more difficult for the K-means algorithm to cluster the snippets correctly. This is aligned with the performance of the code clustering task (Table I). Also, we observe that some points marked in the same color (e.g., red) are somewhat far away from each other even in the vectors from InferCode, while they are supposed to be close according to the ground truth. This could indicate further improvement to Infercode can be made in future work.

B. Effect of Textual Information in TBCNN

The original TBCNN in Mou et al. [43] does not include textual information in AST nodes to initialize the node embedding. In our implementation, we include the textual information by fusing it with the node type information through a linear layer. To help understand the effect of such a fusion process, we perform an ablation study by training InferCode with different initialization information on the Java-Large dataset and perform the evaluations on the three unsupervised tasks: code clustering (CC), code clone detection (CCD), and cross-language code-to-code search (CLCS) with the same settings for each of the tasks in Section VI. Table VI shows the results of this study. Using only type or token information will result in worse performance for all three tasks.

TABLE VI
EFFECTS OF DIFFERENT INITIALIZATION METHODS

Task	Dataset	Metric	Initial Information		
			Type	Token	Combine
CC	OJ	ARI	0.57	0.28	**0.70**
CCD	BigCloneBench	P	0.45	0.49	**0.90**
CLCS	Rosetta Stone	MRR	0.18	0.39	**0.57**

C. Alternative Choices to the Pretext Task Labels

There are a few alternatives when we use subtrees as the pseudo labels for the pretext task in InferCode. One can easily replace the subtrees with tokens so that the code vector $\vec{v}$ can predict the tokens of the code snippets (similar to Doc2vec), or one can use all the method names as the pseudo labels and train the $\vec{v}$ to predict the names, similar to Code2vec [8]. In this section, we perform an ablation study to measure how different types of labels can affect performance. As shown in Table VII, the performance using the subtrees as the labels is the best while using tokens as the labels result in the worst performance. Although using the method name can result in reasonable performance, it is still worse than using the subtrees. An explanation for this is that by predicting method names, the model is forced to learn some incorrect patterns due to similar names in the code base that actually refer to different code. For example, Jiang et al. [70] found that a large number code snippets contain similar method names but the actual implementations of the method bodies are different, but their code vectors would be forced to predict the similar method names, thus these vectors will be close in the vector space despite that they should not be. This is a potential reason to make the model trained by predicting method names a worse choice for pretext task than using subtrees.

VIII. DISCUSSION

A. Choice of Encoder

In this section, we want to discuss our choice on the decoder. We choose TBCNN because of its ability to capture structural

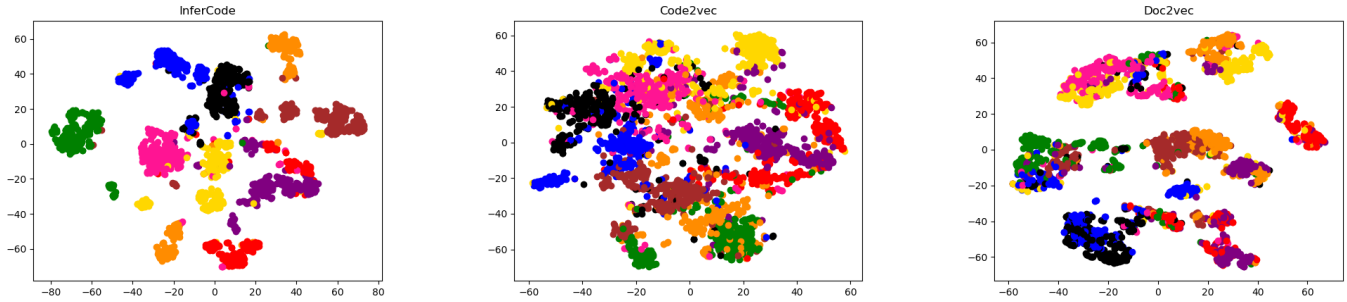

Fig. 6. Visualization of the Code Vectors of the Programs from 9 classes in the OJ Dataset produced by InferCode, Code2vec and Doc2vec

TABLE VII
EFFECTS OF DIFFERENT WAYS TO SET UP LABELS OF THE PRETEXT TASK

Task	Dataset	Metric	Label		
			Token	Method Name	Subtree
CC	OJ	ARI	0.23	0.58	**0.70**
CCD	BigCloneBench	P	0.45	0.81	**0.90**
CLCS	Rosetta Stone	MRR	0.32	0.41	**0.57**

features of code that lie in ASTs and the modification we made to TBCNN can also capture textual information into the model. There are many neural network designs that can be used as a replacement of the TBCNN encoder, such as ASTNN [44], Code2vec [8] or GGNN [49]; however, most of them, especially the graph-based models, are unable to scale and generalize for different programming languages. For example, we can use the path encoder of Code2vec to encode the AST paths into the code vector $\vec{v}$ and infer the subtrees. GGNN is similar, one can pre-train the GGNN over a self-supervised learning task. Although the graph representation proposed by Narayanan et al. [28], Allamanis et al. [49] has been shown to work well on tasks such as supervised clone detection, code summarization, variable name prediction, etc., choosing the suitable edges to be included in the graph representations for such tasks can be time-consuming and not generalizable. LambdaNet [71] is another graph-based model that also contains semantic edges designed specifically for the type prediction task. As such, it is not straightforward to transfer a pre-trained graph learning model through different code learning tasks and it is not easy to scale the graph representation of code into multiple languages. Similar reasons can also be applied for path-based models, such as Code2vec and Code2seq, or execution trace-based models [30]. On the other hand, TBCNN is designed to receive the AST directly with minimal engineering effort to process it. AST is relatively easy to produce accurately for most programming languages given their grammars, thus building a tree-based learning model on top of ASTs implies that we can have a model that is easier to generalize across languages, which is the advantage to choose tree-based models over others. Note that this is not to say that other models do not perform well on the code learning tasks; they can still perform well when training data and time are specially utilized, and they may be used together with each other as the encoder in the self-supervised learning framework to improve the performance for various tasks further. We leave all the exciting explorations for future work.

B. Assumption on Predicting Similar Subtrees with Opposite Meaning

InferCode works on the basis of the key assumption that code snippets containing similar subtrees have the same meanings. There are instances where code snippets can have the opposite meaning even if they have the same subtree, e.g., "$A < B$" vs. "$B < A$." This issue is addressed by modifying the TBCNN to encode the information of the tokens. Note that the original TBCNN Mou et al. [43] only encodes the node type information. With this change, the TBCNN can distinguish both syntactic and semantic information better than the original version, as implied by the results shown in Table VI.

IX. CONCLUSIONS

We have proposed InferCode, a self-supervised learning technique for source code learning on unlabeled data. Along with the document embedding principle that similar documents contain similar words, our working intuition is that similar ASTs should have similar subtrees to predict using a code embedding learnt from the ASTs. We first train a tree-based CNN on large scale datasets, then reuse it as a pre-trained model for the InferCode encoder to map any AST into an embedding vector for downstream tasks, such as code clustering, code clone detection, or code-to-code search. Evaluation of these tasks shows that the embeddings produced by the InferCode encoder outperform the other baselines with significant margins. Furthermore, the weights of the self-supervised pretrained model can be used for subsequent supervised fine-tuning, which outperforms the supervised models trained from a scratch. In the future, we will explore other choices of the encoder and adapt InferCode to other SE tasks such as bug localization, defect prediction, variable name prediction, etc.

ACKNOWLEDGEMENTS

This research is supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant and RISE Lab Operational Fund from SIS at SMU, Singapore MOE AcRF Tier 2 Award No. MOE2019-T2-1-193, Royal Society projects (IES/R1/191138, IES/R3/193175), EPSRC STRIDE project (EP/T017465/1), and Huawei Trustworthy Software Engineering Lab. We also thank the anonymous reviewers for their insightful comments and suggestions, and thank the authors of related work for sharing data.

References

[1] R. Nix and J. Zhang, "Classification of android apps and malware using deep neural networks," in *International Joint Conference on Neural Networks*, May 2017, pp. 1871–1878.

[2] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 3422–3426.

[3] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *40th ICSE*, 2018, pp. 933–944.

[4] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "FaCoY: a code-to-code search engine," in *ICSE*, 2018, pp. 946–957.

[5] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: A neural code search," in *2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, p. 31–41.

[6] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *ICPC*, 2018, pp. 200–210.

[7] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *33rd ASE*, New York, NY, USA, 2018, p. 397–407.

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," in *POPL*, 2019, pp. 40:1–40:29.

[9] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *IEEE QRS*, 2017, pp. 318–328.

[10] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *NeurIPS*, 2019, pp. 10 197–10 207.

[11] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *NeurIPS*, 2018, pp. 2547–2557.

[12] X. Gu, H. Zhang, D. Zhang, and S. Kim, "DeepAM: Migrate apis with multi-modal sequence to sequence learning," in *IJCAI*, 2017, pp. 3675–3681.

[13] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *ICLR*, 2019.

[14] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *29th ISSTA*, 2020, pp. 516–527.

[15] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *27th SANER*, 2020, pp. 261–271.

[16] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *34th ASE*, 2019, pp. 1–12.

[17] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[18] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," *ICML'20*, pp. 1597–1607.

[19] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," *ICML'20*, pp. 10 799–10 808.

[20] C. Doersch and A. Zisserman, "Multi-task self-supervised visual learning," in *ICCV*, 2017, pp. 2051–2060.

[21] A. Kolesnikov, X. Zhai, and L. Beyer, "Revisiting self-supervised visual representation learning," in *CVPR*, 2019, pp. 1920–1929.

[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NeurIPS*, 2013, pp. 3111–3119.

[23] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML*, 2014, pp. 1188–1196.

[24] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *IJCAI*, 2017, pp. 3034–3040.

[25] B. Ingram. (2018) A comparative study of various code embeddings in software semantic matching. https://github.com/waingram/code-embeddings.

[26] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "A doc2vec-based assessment of comments and its application to change-prone method analysis," in *25th APSEC*, 2018, pp. 643–647.

[27] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *arXiv preprint arXiv:1904.03061*, 2019.

[28] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *CoRR*, vol. abs/1707.05005, 2017.

[29] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *15th MSR*, 2018, pp. 542–553.

[30] K. Wang and Z. Su, "Blended, precise semantic program embeddings," in *PLDI'20*, p. 121–134.

[31] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *ICSM*, 2013, pp. 516–519.

[32] Y. Yu, "fast: flattening abstract syntax trees for efficiency," in *ICSE'19*, pp. 278–279.

[33] A. Mahendran, J. Thewlis, and A. Vedaldi, "Cross pixel optical-flow similarity for self-supervised learning," in *Asian Conference on Computer Vision*, 2018, pp. 99–116.

[34] S. Gidaris, P. Singh, and N. Komodakis, "Unsupervised representation learning by predicting image rotations," in

ICLR'18.

[35] R. Zhang, P. Isola, and A. A. Efros, "Colorful image colorization," in *ECCV*, 2016, pp. 649–666.

[36] B. Korbar, D. Tran, and L. Torresani, "Cooperative learning of audio and video models from self-supervised synchronization," in *NeurIPS*, 2018, pp. 7763–7774.

[37] D. Kim, D. Cho, and I. S. Kweon, "Self-supervised video representation learning with space-time cubic puzzles," in *AAAI*, vol. 33, 2019, pp. 8545–8552.

[38] B. Fernando, H. Bilen, E. Gavves, and S. Gould, "Self-supervised video representation learning with odd-one-out networks," in *CVPR*, 2017, pp. 3636–3645.

[39] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, "Skip-thought vectors," in *NeurIPS*, 2015, pp. 3294–3302.

[40] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL'18*, p. 4171–4186.

[41] L. Logeswaran and H. Lee, "An efficient framework for learning sentence representations," in *ICLR'18*, pp. 1–16.

[42] T. Kenter, A. Borisov, and M. de Rijke, "Siamese CBOW: optimizing word embeddings for sentence representations," in *ACL'16*, p. 941–951.

[43] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI*, 2016.

[44] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *41st ICSE*, 2019, pp. 783–794.

[45] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 147, 2018.

[46] R. Gupta, A. Kanade, and S. Shevade, "Neural attribution for semantic bug-localization in student programs," in *NeurIPS*, 2019, pp. 11 861–11 871.

[47] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *7th ICLR*, 2019.

[48] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," in *7th ICLR*, 2019.

[49] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *ICLR*, 2018.

[50] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *ICLR*, Nov. 2016.

[51] L. Mou, H. Peng, G. Li, Y. Xu, L. Zhang, and Z. Jin, "Discriminative neural sentence modeling by tree-based convolution," in *EMNLP*, 2015, pp. 2315–2325.

[52] N. D. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," in *NL4SE@AAAI'18*, 2018.

[53] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *27th ICPC*, 2019, pp. 70–80.

[54] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, 2002.

[55] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *26th ESEC/FSE*, 2018, pp. 354–365.

[56] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[57] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR'15*.

[58] B. D. Q. Nghi, Y. Yu, and L. Jiang, "Bilateral dependency neural networks for cross-language algorithm classification," in *SANER'19*, pp. 422–433.

[59] J. M. Santos and M. Embrechts, "On the use of the adjusted rand index as a metric for evaluating supervised classification," in *International conference on artificial neural networks*, 2009, pp. 175–184.

[60] F. Hill, K. Cho, and A. Korhonen, "Learning distributed representations of sentences from unlabelled data," in *NAACL'16*, p. 1367–1377.

[61] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th ICSE*, 2007, pp. 96–105.

[62] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *38th ICSE*, 2016, pp. 1157–1168.

[63] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *31st ASE*, 2016, pp. 87–98.

[64] L. Wang, J. Lin, and D. Metzler, "A cascade ranking model for efficient ranked retrieval," in *34th SIGIR*, 2011, pp. 105–114.

[65] S. Wan, Y. Lan, J. Guo, J. Xu, L. Pang, and X. Cheng, "A deep architecture for semantic matching with multiple positional sentence representations." in *AAAI*, vol. 16, 2016, pp. 2835–2841.

[66] I. Vulić and M.-F. Moens, "Monolingual and cross-lingual information retrieval models based on (bilingual) word embeddings," in *38th SIGIR*, 2015, pp. 363–372.

[67] Y. Kim, "Convolutional neural networks for sentence classification," p. 1746–1751.

[68] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, 1997.

[69] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[70] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: how far are we," in *34th ASE*, 2019, pp. 602–614.

[71] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "Lambdanet: Probabilistic type inference using graph neural networks," in *ICLR'20*.